

E09 Variable Elimination

16337102 Zilin Huang

November 5, 2018

Contents

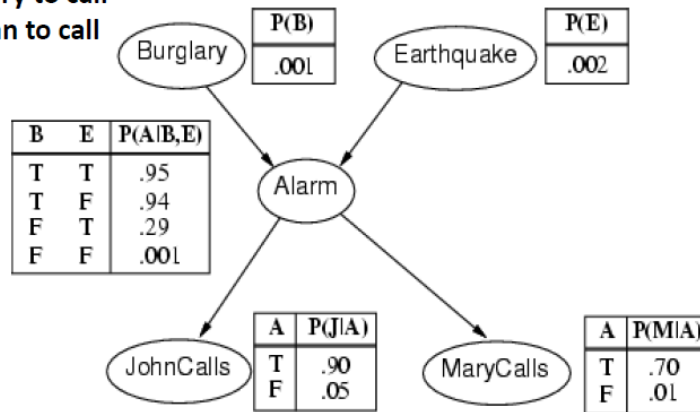
1	VE	2
2	Task	5
3	Codes	5
4	Results	12

1 VE

The burglary example is described as following:

- A burglary can set the alarm off
- An earthquake can set the alarm off
- The alarm can cause Mary to call
- The alarm can cause John to call

Note that these tables only provide the probability that X_i is true.
(E.g., $\Pr(A \text{ is true} | B, E)$)
The probability that X_i is false is 1- these values



```
P(Alarm) =
0.002516442

P(J&&~M) =
0.050054875461

P(A | J&&~M) =
0.0135738893313

P(B | A) =
0.373551228282

P(B | J&&~M) =
0.0051298581334

P(J&&~M | ~B) =
0.049847949
```

Here is a VE template for you to solve the burglary example:

```
class VariableElimination:
    @staticmethod
    def inference(factorList, queryVariables,
        orderedListOfHiddenVariables, evidenceList):
        for ev in evidenceList:
            #Your code here
        for var in orderedListOfHiddenVariables:
            #Your code here
        print "RESULT:"
```

```

        res = factorList[0]
        for factor in factorList[1:]:
            res = res.multiply(factor)
        total = sum(res.cpt.values())
        res.cpt = {k: v/total for k, v in res.cpt.items()}
        res.printInf()

    @staticmethod
    def printFactors(factorList):
        for factor in factorList:
            factor.printInf()

class Util:
    @staticmethod
    def to_binary(num, len):
        return format(num, '0' + str(len) + 'b')

class Node:
    def __init__(self, name, var_list):
        self.name = name
        self.varList = var_list
        self.cpt = {}

    def setCpt(self, cpt):
        self.cpt = cpt

    def printInf(self):
        print "Name_=" + self.name
        print "_vars_" + str(self.varList)
        for key in self.cpt:
            print "___key:_ " + key + "_val:_ " + str(self.cpt[key])
        print ""

    def multiply(self, factor):
        """function that multiplies with another factor"""
        #Your code here
        new_node = Node("f" + str(newList), newList)
        new_node.setCpt(new_cpt)
        return new_node

```

```

def sumout(self, variable):
    """function that sums out a variable given a factor"""
    #Your code here
    new_node = Node("f" + str(new_var_list), new_var_list)
    new_node.setCpt(new_cpt)
    return new_node

def restrict(self, variable, value):
    """function that restricts a variable to some value
    in a given factor"""
    #Your code here
    new_node = Node("f" + str(new_var_list), new_var_list)
    new_node.setCpt(new_cpt)
    return new_node

# create nodes for Bayes Net
B = Node("B", ["B"])
E = Node("E", ["E"])
A = Node("A", ["A", "B", "E"])
J = Node("J", ["J", "A"])
M = Node("M", ["M", "A"])

# Generate cpt for each node
B.setCpt({'0': 0.999, '1': 0.001})
E.setCpt({'0': 0.998, '1': 0.002})
A.setCpt({'111': 0.95, '011': 0.05, '110':0.94, '010':0.06,
'101':0.29, '001':0.71, '100':0.001, '000':0.999})
J.setCpt({'11': 0.9, '01': 0.1, '10': 0.05, '00': 0.95})
M.setCpt({'11': 0.7, '01': 0.3, '10': 0.01, '00': 0.99})

print "P(A) ⊥*****"
VariableElimination.inference([B,E,A,J,M], ['A'], ['B', 'E', 'J', 'M'], {})

print "P(B ⊥ J ⊥ M) ⊥*****"
VariableElimination.inference([B,E,A,J,M], ['B'], ['E', 'A'], {'J':1, 'M':0})

```

2 Task

- You should implement 4 functions: `inference`, `multiply`, `sumout` and `restrict`. You can turn to Figure 1 and Figure 2 for help.
- Please hand in a file named `E09_YourNumber.pdf`, and send it to `ai_2018@foxmail.com`

The VE Algorithm

Given a Bayes Net with CPTs F , query variable Q , evidence variables E (observed to have values e), and remaining variables Z . Compute $\Pr(Q|E)$

- Replace each factor $f \in F$ that mentions a variable(s) in E with its restriction $f_{E=e}$ (this might yield a "constant" factor)
- For each Z_j —in the order given—eliminate $Z_j \in Z$ as follows:
 - Let f_1, f_2, \dots, f_k be the factors in F that include Z_j
 - Compute new factor $g_j = \sum_{Z_j} f_1 \times f_2 \times \dots \times f_k$
 - Remove the factors f_i from F and add new factor g_j to F
- The remaining factors refer only to the query variable Q . Take their product and normalize to produce $\Pr(Q|E)$.

The Product of Two Factors

- Let $f(\underline{X}, \underline{Y})$ & $g(\underline{Y}, \underline{Z})$ be two factors with variables \underline{Y} in common
- The **product** of f and g , denoted $h = f \times g$ (or sometimes just $h = fg$), is defined:

$$h(\underline{X}, \underline{Y}, \underline{Z}) = f(\underline{X}, \underline{Y}) \times g(\underline{Y}, \underline{Z})$$

f(A,B)		g(B,C)		h(A,B,C)			
ab	0.9	bc	0.7	abc	0.63	ab~c	0.27
a~b	0.1	b~c	0.3	a~bc	0.08	a~b~c	0.02
~ab	0.4	~bc	0.8	~abc	0.28	~ab~c	0.12
~a~b	0.6	~b~c	0.2	~a~bc	0.48	~a~b~c	0.12

Figure 1: VE and Product

Summing a Variable Out of a Factor

- Let $f(X, \underline{Y})$ be a factor with variable X (\underline{Y} is a set)
- We **sum out** variable X from f to produce a new factor $h = \sum_X f$, which is defined:

$$h(\underline{Y}) = \sum_{X \in \text{Dom}(X)} f(X, \underline{Y})$$

f(A,B)		h(B)	
ab	0.9	b	1.3
a~b	0.1	~b	0.7
~ab	0.4		
~a~b	0.6		

No error in the table. Here $f(A, B)$ is not $P(A, B)$, but $P(B|A)$.

Restricting a Factor

- Let $f(X, \underline{Y})$ be a factor with variable X (\underline{Y} is a set)
- We **restrict** factor f to $X=a$ by setting X to the value a and "deleting" incompatible elements of f 's domain. Define $h = f_{X=a}$ as: $h(\underline{Y}) = f(a, \underline{Y})$

f(A,B)		h(B) = f_{A=a}	
ab	0.9	b	0.9
a~b	0.1	~b	0.1
~ab	0.4		
~a~b	0.6		

Figure 2: Sumout and Restrict

3 Codes

```
import copy

def compare(list1, list2):
    num = len(list1)
```

```

if num == len(list2):
    for i in range(num):
        if list1[i] != list2[i]:
            return False
    return True
return False

def VE(factorList, queryVariables, orderedList, evidenceList):
    #restrict
    for ele, value in evidenceList.items():
        for factor in factorList:
            if ele in factor.varList:
                new_node = factor.restrict(ele, value)
                factorList.remove(factor)
                factorList.append(new_node)

    #eliminate
    for var in orderedList:
        #find factor with var
        mid_list = []
        for factor in factorList:
            if var in factor.varList:
                mid_list.append(factor)
        for factor in mid_list:
            factorList.remove(factor)

    while len(mid_list) != 1:
        for i in range(len(mid_list)):
            if i >= len(mid_list):
                break
            ele = mid_list[i].varList[-1]
            for j in range(len(mid_list)):

```

```

        if j >= len(mid_list):
            break
        if i != j:
            ele_ = mid_list[j].varList[0]
            if ele == ele_:
                mid_list[i]=mid_list[i].multiply(mid_list[j])
            del mid_list[j]

    fir = mid_list[0]
    fir = fir.sumout(var)
    factorList.append(fir)

for factor in factorList:
    if len(factor.varList) == 0:
        factorList.remove(factor)

tar = queryVariables[0]
mid_list = []
for factor in factorList:
    if tar not in factor.varList:
        mid_list.append(factor)
for factor in mid_list:
    factorList.remove(factor)
res = factorList[0]
for factor in factorList[1:]:
    res = res.multiply(factor)
#normalize
total = sum(res.cpt.values())
res.cpt = {k: v/total for k, v in res.cpt.items()}
return res

```

```

class Node:
    def __init__(self, name, var_list):
        global number
        self.name = name
        self.varList = var_list
        self.cpt = {}

    def setCpt(self, cpt):
        self.cpt = cpt

    def printInf(self):
        print("Name_= " + self.name)
        print("_vars_" + str(self.varList))
        for key in self.cpt:
            print("_key_" + key + "_val_:_" + str(self.cpt[key]))
        print(' ')

    def multiply(self, factor): # factor is node
        new_cpt = {}
        newList = []
        if self.varList[-1] == factor.varList[0]: # f(a, b) X g(b, c)
            # new variable list, order is important
            del factor.varList[0]
            newList = self.varList + factor.varList
            for key, value in self.cpt.items():
                for key_, value_ in factor.cpt.items():
                    new_key = list(key)
                    new_key_ = list(key_)
                    if new_key[-1] == new_key_[0]:
                        del new_key_[0]
                        result_key = new_key + new_key_ # new key
                        result_value = value * value_ # new value
                        result_key = "".join(result_key) #list to string

```



```

        if not new_cpt.__contains__(result_key):
            new_cpt[result_key] = result_value
new_node = Node("f" + str(newList), newList)
new_node.setCpt(new_cpt)
return new_node

def sumout(self, variable):
    # find the position of variable
    position = self.varList.index(variable)
    new_var_list = copy.deepcopy(self.varList)
    new_var_list.remove(variable) # delete variable
    new_cpt = {}
    for key, value in self.cpt.items(): # modify the cpt
        for key_, value_ in self.cpt.items():
            new_key = list(key)
            new_key_ = list(key_)
            if not compare(new_key, new_key_): #two keys are different
                del new_key[position]
                del new_key_[position]
            if compare(new_key, new_key_):
                new_key = "".join(new_key)
                value = value + value_
                if not new_cpt.__contains__(new_key):
                    new_cpt[new_key] = value
                break
    new_node = Node("f" + str(new_var_list), new_var_list)
    new_node.setCpt(new_cpt)
    return new_node

def restrict(self, variable, value):
    position = self.varList.index(variable)
    new_var_list = copy.deepcopy(self.varList)
    new_var_list.remove(variable) # delete variable

```

```

        new_cpt = {}
        for key, value_ in self.cpt.items(): # modify the cpt
            new_key = list(key)
            if int(new_key[position]) == value: # item in new cpt
                del new_key[position]
                key_ = "".join(new_key)
                new_cpt[key_] = value_ # new cpt
        new_node = Node("f" + str(new_var_list), new_var_list)
        new_node.setCpt(new_cpt)
        return new_node

def initial():
    B = Node("B", ["B"])
    E = Node("E", ["E"])
    A = Node("A", ["B", "E", "A"])
    J = Node("J", ["A", "J"])
    M = Node("M", ["A", "M"])

    B.setCpt({'1': 0.001, '0': 0.999})
    E.setCpt({'1': 0.002, '0': 0.998})
    A.setCpt({'111': 0.95, '110': 0.05, '101': 0.94, '100': 0.06,
              '011': 0.29, '010': 0.71, '001': 0.001, '000': 0.999})
    J.setCpt({'11': 0.9, '10': 0.1, '01': 0.05, '00': 0.95})
    M.setCpt({'11': 0.7, '10': 0.3, '01': 0.01, '00': 0.99})
    return [B, E, A, J, M]

print("P(A)  $\models$ ")
factorList = initial()
res = VE(factorList, ['A'], ['B', 'E', 'J', 'M'], {})
p_A = res.cpt["1"]
p_a = res.cpt["0"]

```

```

print(p_A)

print("P(B_⊥ | ⊥J~M) ⊥=")
factorList = initial()
res = VE(factorList, ['B'], ['A', 'E'], {'J': 1, 'M': 0})
p_B_Jm = res.cpt["1"]
print(p_B_Jm)

print("P(A_⊥ | ⊥J~M) ⊥=")
factorList = initial()
res = VE(factorList, ['A'], ['B', 'E'], {'J': 1, 'M': 0})
p_A_Jm = res.cpt["1"]
print(p_A_Jm)

print("P(B_⊥ | ⊥A) ⊥=")
factorList = initial()
res = VE(factorList, ['B'], ['E', 'J', 'M'], {'A': 1})
p_B_A = res.cpt["1"]
print(p_B_A)

factorList = initial()
res = VE(factorList, ['J'], ['B', 'E', 'A'], {'M': 0})
p_J_m = res.cpt["1"]

factorList = initial()
res = VE(factorList, ['M'], ['B', 'E', 'J'], {'A': 1})
p_M_A = res.cpt["1"]
p_m_A = res.cpt["0"]

factorList = initial()
res = VE(factorList, ['M'], ['B', 'E', 'J'], {'A': 0})
p_M_a = res.cpt["1"]

```

```

p_m_a = res.cpt["0"]

p_m = p_m_A * p_A + p_m_a * p_a
p_Jm = p_m * p_J_m
print ("P(J&&M) =")
print (p_Jm)

p_B = 0.001
p_b = 1 - p_B
p_b_Jm = 1 - p_B_Jm
p_Jm_b = p_b_Jm * p_Jm / p_b
print ( 'P(~B|J&&~B) =')
print (p_Jm_b)

```

4 Results

```

P(A) =
0.0025164420000000002
P(B | J~M) =
0.0051298581334013
P(A | J~M) =
0.013573889331307633
P(B | A) =
0.373551228281836
P(J&&~M) =
0.050054875461
P(~B|J&&~B) =
0.049847949

```